基于动态适应度函数的模糊测试技术研究 *

邓一杰，刘克胜，朱凯龙，常　超

(国防科技大学 电子对抗学院，合肥 230031)

摘　要：模糊测试是一种有效的自动化漏洞挖掘技术，主流模糊测试技术采用遗传算法生成测试用例，存在早熟现象，导致路径覆盖率不足。针对该问题，提出一种基于动态适应度函数的模糊测试方法。综合考虑了种子新度和路径深度因素，设计了根据测试阶段不同而动态变化的适应度函数，实现了基于动态适应度函数的模糊测试工具 DynFuzzer。在 BegBunch 和 CGC 提供的测试集上进行实验，结果表明与现有模糊测试工具相比，DynFuzzer 路径覆盖率提高了 40%，多发现了 10%的 bug。基于动态适应度函数的模糊测试方法能有效克服早熟问题，提高路径覆盖率，发现更多的 bug。

关键词：模糊测试；遗传算法；动态适应度函数；DynFuzzer；路径覆盖率

中图分类号：TP311.5　　doi: 10.3969/j.issn.1001-3695.2018.04.0266

Research on fuzzing technique based on dynamic fitness function

Deng Yijie, Liu Kesheng, Zhu Kailong, Chang Chao

(*National University of Defense Technology*, *Electronic Engineering Institute*, *HeFei* 230000, *China*)

Abstract: Fuzzing is an effective technique for automatically mining vulnerabilities. The mainstream fuzzing technique uses genetic algorithm to generate cases for testing, but almost there exists a premature phenomenon, which leads to lower ratio of path coverage. Given this problem, this paper proposed a fuzzing test method based on dynamic fitness function. Considering the newness of seed and the depth of the path, designed an improved fitness function of dynamic change with different test phases, by which, implemented the fuzzing testing tool–DynFuzzer. On the test set provided by BegBunch and CGC, devised a experiment. The results show that compared with the existing fuzzing test tools, the DynFuzzer path coverage is 40% higher and 10% more bugs are found. The fuzzing test method based on dynamic fitness function can overcome the problem of prematurity, improve path coverage and find more bugs.

Key words: fuzzing; genetic algorithm; dynamic fitness function; DynFuzzer; path coverage

0　引言

模糊测试是一种有效的漏洞挖掘技术，具有高度自动化、可扩展性强、可适用于大型程序等优势，被广泛应用于软件测试领域错误!未找到引用源。据统计，微软产品发布前有 20%~25%的安全漏洞是通过模糊测试技术发现的错误!未找到引用源。因此，设计一种高效的模糊测试方法在软件安全领域具有重要意义。

根据测试用例生成方法不同，可以将模糊测试分为：基于格式分析和程序理解相结合的和基于静态分析与动态测试相结合的错误!未找到引用源。，前者的代表工具有 SPIKE、Peach 错误!未找到引用源。等，该方法的优点是执行效率高、应用范围广、通用性强，缺点是仍然需要大量的人工参与来进行多种知识的获取错误!未找到引用源。；后者的代表工具有 Buzz-Fuzz 错误!未找到引用源。等，该方法的优点是代码覆盖率较高，缺点是无法克服路径爆炸的问题，也无法解决高强度的程序检查。2016 年结合启发式算法的模糊测试工具 AFL 因为其简单、高效的优点成为了工业化漏洞挖掘最常用的模糊工具之一错误!未找到引用源。。

AFL 采用以路径覆盖率为导向的遗传算法进行模糊测试，相比其他模糊测试方法，提高了路径覆盖率错误!未找到引用源。。但是，AFL 的遗传算法采用固定的适应度函数，而没有充分利用在测试过程中动态获得的程序路径信息，导致种群基因收敛过早，测试后期难以发现新的路径，路径覆盖率不足。

针对上述问题，本文提出了一种基于动态适应度函数的模糊测试方法，适应度函数的设计综合考虑路径的新度和深度两个因素，在模糊测试的不同阶段赋予其不同的权值以得到更高的路径覆盖率。

收稿日期：2018-04-10；**修回日期：**2018-06-08　　**基金项目：**国家重点研发计划重点专项资助项目（2017YFB0802905）

作者简介：邓一杰（1996-），男，湖北荆州人，硕士研究生，主要研究方向为网络安全、模糊测试（bit_eurus@163.com）；刘克胜（1968-），男，安徽桐城人，教授，博士研究生，主要研究方向为漏洞挖掘和分析利用；朱凯龙（1991-），男，河南开封人，硕士研究生，主要研究方向为漏洞挖掘分析、符号执行；常超（1989-），男，河南焦作人，博士研究生，主要研究方向为符号执行、软件安全.

1 问题分析

1.1 AFL 的测试用例生成方法

AFL 采用遗传算法生成测试用例错误!未找到引用源。。首先，用户提供初始测试用例集，然后，以路径覆盖作为导向，采用变异操作产生下一代测试用例，将能产生新路径的用例保留为种子，循环上述操作,通过不断覆盖程序的执行路径来发现 bug 错误!未找到引用源。。

AFL 中种子从测试用例中选择而来，测试用例又由种子变异产生，如果该测试用例在执行过程中产生了新的路径则将其加入种子集中生成下一代种群，这样能建立种子和路径的一一对应关系错误!未找到引用源。，这也表示 AFL 的种群数量是只增不减。在种子执行过程中 AFL 还有其独特的 favorite 策略，对于每条路径片段，将到达该片段速度最快或输入体积最小的种子标记为 favorite 错误!未找到引用源。，被标记为 favorite 的种子在选择时会拥有更高的概率被选中。

在种子被选中后，每个种子会变异生成一定数量的测试用例，生成测试用例的数量由种子的执行时间、路径片段覆盖率和产生新输入所花的时间共同决定，被标记为 favorite 的种子每次被选择时会生成几千个测试用例，其他种子被选中后只产生几十个测试用例。AFL 特殊的种子选择和能量分配策略提升了模糊测试的效率但也导致了其种群基因收敛过早，代码覆盖率不足。

1.2 AFL 实例分析

使用 AFL 对 CGC 中 b64_encode_1 赛题进行模糊测试，测试时间为 2 小时。AFL 进行了约 2000 次循环，每个种子被选择次数分布如图 1 所示。

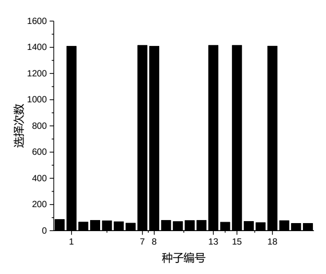


图 1 AFL 的种子选择情况

横坐标表示种子的编号，纵坐标表示种子被选择的次数。可以发现编号为 1、7、8、13、15、18 的种子被选择的次数远高于其他的种子，这些种子也是被标记为 favorite 的种子。

在与上一个实验相同的实验环境中统计每个种子生成的测试用例数量并计算得到种子被选中时平均生成测试用例的数量，具体如图 2 所示。

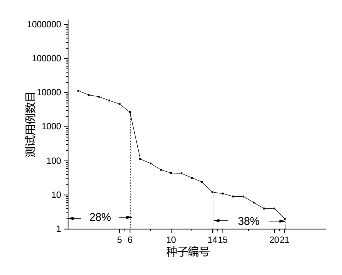


图 2 AFL 的种子生成测试例情况

横坐标表示对所有种子按照生成测试例数进行降序排序的编号，纵坐标表示每个种子在一个 Fuzz 循环中生成的测试用例的数量即路径被执行的次数。可以发现 AFL 在测试例生成上有如下几个特点：a)有 38%的种子只生成了极少的测试用例；b)有 28%的种子生成了绝大多数的测试用例。

可以看出 AFL 的大部分时间都在执行 favorite 种子，非 favorite 的种子执行时间占不到 1%，但是 favorite 的判定是根据种子的执行速度和体积大小决定的，与是否有可能产生新路径并没有直接关系，以上原因导致 AFL 的种群基因收敛过早，路径覆盖率不足。

2 基于路径状态的动态适应度函数

为了能有效地把模糊测试的资源集中到缺陷多发区，取得更高的路径覆盖率，本文提出了一种基于动态适应度函数的模糊测试方法。方法结合了两个适应度函数：新度优先适应度函数和深度优先适应度函数,采用了根据测试阶段不同而动态变化的适应度函数。

2.1 新度适应度函数

新度适应度函数的目的是为了使程序的控制流图尽可能完整并发现一些浅层的 bug。根据实验和漏洞挖掘经验，本文总结提出两条模糊测试前期的启发式规则。

启发式规则 1 一个种子发现的路径片段越新，则其后代在下一轮测试中发现新路径的概率越大。

启发式规则 2 一个种子所执行路径上的片段越多，其后代在下一轮测试中发现新路径的概率越大。

根据上述启发式规则，定义种子的新度适应度等于其对应执行路径上所有路径片段的新度和

$$Fitness_nfs_value(seed)=\sum_{i=1}^{n}2^{cyc(i)}$$

其中:i 为种子对应执行路径上的路径片段编号，cyc(i)为路径片段 i 第一次被执行的循环轮次，$2^{cyc(i)}$ 表示路径片段 i 的新度。将时间对适应度的影响定义为指数级别，提高了发现新路径片段的种子被选中的优先级。

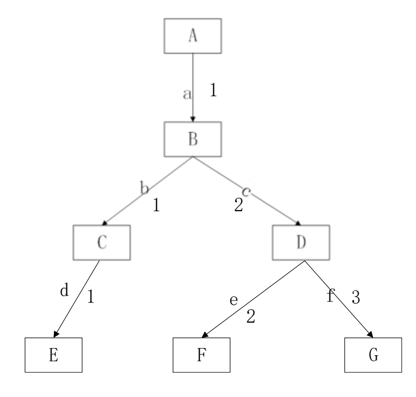


图 3 新度适应度计算

通过图 3 中的实例解释测试用例新度的计算方法。

图 3 表示一个程序的部分控制流，路径片段上的数字表示该路径片段被发现的轮次。

假设有 c1、c2、c3 三个种子，执行路径分别为 a-b-d、a-c-e、a-c-f 可以计算三个种子的新度适应度：

$$Fitness_nfs_value(c1)=2^1+2^1+2^1=6$$

$$Fitness_nfs_value(c2)=2^1+2^2+2^2=10$$

$$Fitness_nfs_value(c3)=2^1+2^2+2^3=14$$

就得到了三个种子的新度适应度。

2.2　深度适应度函数

深度适应度函数的目的是为了从程序的路径中选出危险性较高的路径，有针对性的对危险路径进行模糊测试以发现深层的 bug。根据实验和漏洞挖掘经验，本文总结提出一条模糊测试后期的启发式规则。

启发式规则 3　种子对应的路径越深，该条路径上越有可能存在 bug。

基于上述启发式规则，在模糊测试后期以路径的深度作为适应度函数的优先影响因子，以求提高发现深层 bug 的概率。

2.2.1 控制流到马尔可夫链的映射

马尔可夫链是概率学中的一种离散事件随机过程，该过程中的状态转换满足如下性质：过去状态对于将来状态的预测是无影响的，将来的状态只与现在的状态有关错误!未找到引用源。本文研究的目标是以 C 语言等结构化程序语言编写的二进制程序，针对结构化的程序，可以将程序的每个基本块视为一种状态，之前经过的基本块对于下一步可能到达的基本块是无影响的，下一步可能到达的基本块只与现在所在的基本块有关系，所以程序的执行流程是满足马尔可夫性质的，这种随机过程可以抽象为一条马尔可夫链[14]。

把程序基本块视为状态后，路径就可视为状态转移过程，对应的每条路径的概率就是状态转移概率，这个概率由统计得出，AFL 在程序的每个分支点处插桩了一些轻型代码以获得路径信息[14]，本方法利用这些代码记录了在分支点处选择不同分支的测试用例数，分支用例数所占总用例数的比例定义为该路径分支的概率，具体计算方法如图 4 所示。

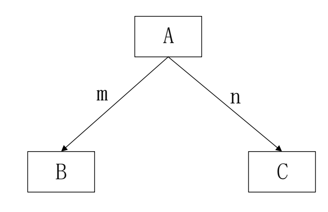


图 4　路径分支概率

假设在一次 Fuzz 过程后，有 m 个测试用例经过路径分支 AB，n 个测试用例经过分支 AC，则

$$p(AB)=\frac{m}{m+n},\ p(AC)=\frac{n}{m+n}\text{。}$$

2.2.2 路径的权重计算

本方法通过计算每条路径的权重获得种子的深度适应度，每个种子的深度适应度由其对应的路径上的基本块权重相加得到，基本块概率由路径状态转移概率求得，公式如下：

$$p(S)=\sum_{D\in u(S)}p(D)*p(DS)$$

其中：S 为基本块，u(S)为能通过一步状态转移到达 S 的基本块的集合，DS 表示基本块 D 到基本块 S 的路径，p(Dot)和 p(edge)分别表示基本块的概率和路径片段的概率。

每个基本块的权重定义为其状态概率的倒数，S 基本块的权重为

$$w(S)=\frac{1}{p(S)}$$

若某个种子 seed 对应路径经过的基本块集合为 block(seed) 则该路径对应种子的深度适应度为

$$Fitness_dfs_value(seed)=\sum_{Dot\in block(seed)}w(Dot)$$

需要特别指出的是，模糊测试过程中会不断产生新的路径，所以程序的控制流图也会动态变化，每经过一次 Fuzz 循环就更新一次控制流图，然后依靠新的控制流图计算新的种子适应度并进行新一轮的 Fuzz。图 5 展示了一次完整的 Fuzz 过程路径权值计算。

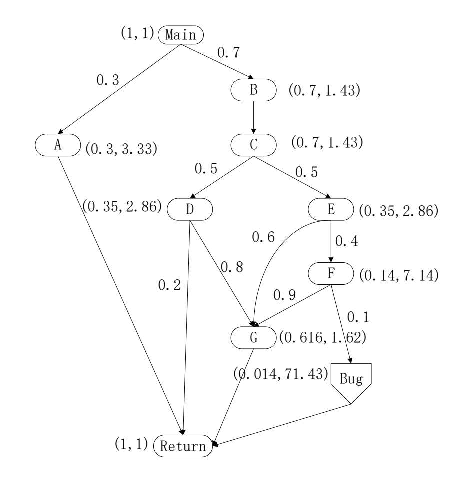


图 5　路径适应度计算

图 5 表示程序的控制流，基本块旁的值(a,b)，a 表示基本块的概率，b 表示基本块的权值，基本块之间的数字表示基本块间的转移概率。假设现在有一个种子 c4 对应着 Main-B-C-E-F-Bug-Return 路径，另一个种子 c5 对应着 Main-B-C-D-G-Return 路径，分别依靠现有控制流计算两个种子的深度适应度，计算方法如下:

先基于现在控制流图计算出的每个基本块概率和权值：

$$p(B)=1*0.7=0.7$$

$$w(B)=\frac{1}{p(B)}=2$$

$$p(G)=0.35*0.8+0.35*0.6+0.14*0.9=0.616$$

$$w(G) = \frac{1}{p(G)} = 1.62$$

再把路径包含的基本块权值相加即可得到对应种子的深度适应度，相加的时候需要去掉程序的起始块和结束块：

$$Fitness_dfs_value(c4) = w(B) + w(C) + w(E) + w(F) + w(Bug) = 84.29$$

$$Fitness_dfs_value(c5) = w(B) + w(C) + w(D) + w(G) = 7.34$$

得出两个种子的深度适应度。

2.3 动态适应度函数

基于上文提出的两种适应度函数计算方法，本方法在不同的时期分别赋予其不同的权值实现了动态的适应度函数设计，前期新度适应度占权重较高，后期深度适应度占权重较高，具体设计方法如下：

为了避免新度适应度和深度适应度相互影响，首先将其均标准化处理

$$Fitness_nfs(seed) = \frac{Fitness_nfs_value(seed)}{\sum_{j \in set} Fitness_nfs_value(j)}$$

$$Fitness_dfs(seed) = \frac{Fitness_dfs_value(seed)}{\sum_{j \in set} Fitness_dfs_value(j)}$$

set 表示该次循环中的种子集。

然后将标准化后的两个适应度函数赋予不同的权重后相加：

$$Fitness = \begin{cases} (1-\log_a(cyc))Fitness_nfs + \log_a(cyc)Fitness_dfs & cyc \le \sqrt{a} \\ \frac{b}{cyc}Fitness_nfs + (1-\frac{b}{cyc})Fitness_dfs & cyc > 2b \end{cases}$$

为了让适应度函数连续，参数 a 和 b 满足条件 $\sqrt{a} = 2b = \text{rush}$，rush 的值根据不同程序规模可自定义，默认为 20，rush 是一个适应度阈值，表示在小于 rush 的轮次中新度适应度的权重大于深度的权重，在大于 rush 的轮次中新度适应度的权重小于深度的权重。这种设计方法通过动态改变遗传算法的适应度函数避免种群的收敛并有效的提高路径覆盖率和发现深层 bug 的概率。

这种基于动态适应度函数的模糊测试方法可以有效的发现程序的完整路径并将模糊测试的时间和计算资源分配给更可能存在 bug 的区域，有利于发现代码更深层次的缺陷，不足之处是在前期发现 crash 的速率方面略有下降。

3 DynFuzzer 系统设计与实现

根据上文提出的基于动态适应度函数的模糊测试方法设计实现了模糊测试工具 DynFuzzer，本节对 DynFuzzer 使用的遗传算法和系统架构进行介绍。

3.1 种子选择和能量分配算法

DynFuzzer 通过种子的动态适应度值挑选进行 Fuzz 的种子并给其分配能量，在本文中定义能量为种子变异产生测试用例的数量。在计算得到种子集中种子的动态适应度后，采用轮盘赌算法选择种子进行 Fuzz，每个种子都有一定的概率被选中，选择种子 i 的概率为

$$Pro(i) = \frac{Fitness(i)}{Fit}$$

其中：

$$Fit = \sum_{i=1}^{n} Fitness(i)$$

选择种子后进入能量分配阶段，适应度高的种子能量大，将产生更多的测试用例，在 Fuzz 开始前针对每个种子定义一个最大测试用例数。

$$MAX_case(i) = 2^{split(i)}$$

split(i)为种子 i 对应路径所含的路径片段数，种子 i 最终被分配的能量为

$$power(i) = \begin{cases} 10^2 \\ Pro(i) * MAX_case(i) & 10^2 < power(i) < 10^5 \\ 10^5 \end{cases}$$

该公式表明给种子分配的能量是有边界的，最小产生 100 个测试用例，最多产生 10^5 个。

3.2 系统架构

DynFuzzer 系统分为两个模块：代理模块和监控模块，模块设计如图 6 所示。

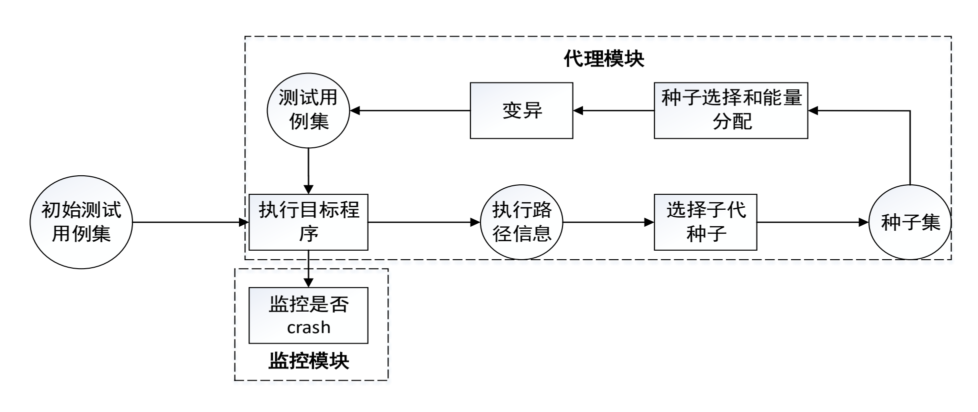


图 6　DynFuzzer 模块架构

代理模块：模糊测试的主模块，负责从种子集中选出进行 Fuzz 的种子并将其变异生成测试用例；调用目标程序执行测试用例；选择执行了新路径的测试用例将其加入到下一代种子集中。

监控模块：负责监控测试用例在执行过程中是否产生 crash，并记录产生 crash 时的现场如各寄存器的值和堆栈的值等。

系统的运行流程如下：

a)用户输入初始测试用例集并执行；

b)记录程序的执行路径信息，若产生 crash 则调用监控模块记录程序详细执行信息，然后选择执行了新路径的测试用例加入下一代种子集；

c)根据第 b)步的路径信息计算出种子的适应度，根据种子的适应度进行种子选择和能量分配操作；

d)将第 c)步中选出来的种子变异生成测试用例集并调用目标程序执行，转到步骤 b)；

Fuzz 循环会一直持续，直到用户手动结束或者当超过一定时间没有产生新的路径后停止测试。

4　实验及分析

4.1　BegBunch

针对本论文设计的 DynFuzzer 采用 BegBunch[15]作为实验数据集，BegBunch 是一个人工构造的包含 67 个 crash 的 bug 测试集。本实验的运行环境为 64 位的 Ubuntu 14.04、8 个核心 CPU、内存为 12 GB，运行时间为 6 h，AFL 和 DynFuzzer 在不同时间点发现的路径数和 crash 数统计如图 7 和 8 所示。

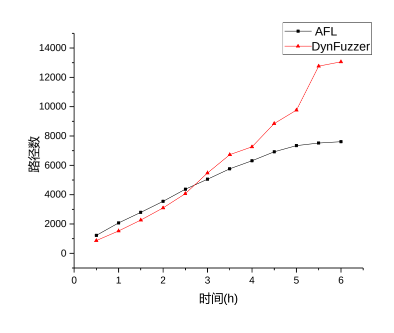


图 7　路径对比图

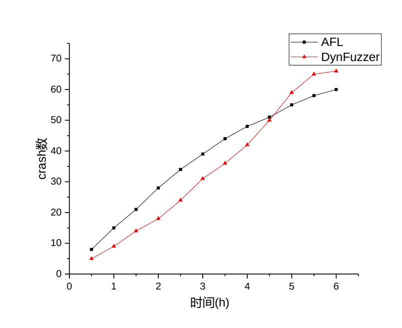


图 8　缺陷对比图

横坐标表示 Fuzz 的时间，图 7 和 8 的纵坐标分别表示发现的路径数和 crash 数，通过实验结果可以发现，DynFuzzer 在前期发现新路径的能力上略弱于 AFL，但是当经过一定次数循环后，DynFuzzer 最终发现了 13 058 条路径，AFL 只发现了 7 613 条路径，DynFuzzer 比 AFL 多发现了 40%的路径；在产生 crash 方面，DynFuzzer 的能力在不同时间阶段能保持相对稳定，后期的 Fuzz 能力与前期相差不大，最终发现了 66 个 crash；而 AFL 的 Fuzz 能力明显随着时间的推移越来越差，这也说明了 AFL 后期 Fuzz 能力不足的问题，最终只发现了 60 个 crash，采用了动态适应度函数的 DynFuzzer 比 AFL 多发现了 10%的 crash。

4.2　CGC 赛题

收集 CGC(cyber grand challenge)比赛中用到的赛题作为本次实验测试集，由于 CGC 赛题的难度不一，对于一些较简单的题 AFL 和 DynFuzzer 都能比较快的发现 crash，太难的题都无法发现 crash，看不出性能上的差别，因此只统计了中等难度的、Fuzz 时间较长的测试用例测试情况，这次实验运行环境与 4.1 节相同，不同点是两种工具均让其运行 12 h 后统计 crash 情况，结果如图 9 所示。

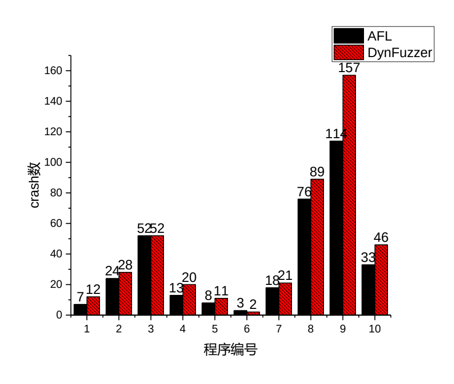


图 9　基于 CGC 的实验测试

横坐标表示 CGC 赛题的编号，纵坐标表示模糊测试结束后发现的 crash 数，通过实验发现，DynFuzzer 总共发现了 438 个 crash，而 AFL 发现了总共 348 个 crash，DynFuzzer 比 AFL 多发现 26%的 crash，考虑到统计的 CGC 赛题都是中等难度的测试题，真实的测试集中还包含 10 道简单的测试题，对这些题 AFL 和 DynFuzzer 均能发现所有的 crash，还有 5 道对于难度系数很高的 CTF 赛题，AFL 和 DynFuzzer 都无法发现 crash，综合这些题目对比较实验的影响后，计算得到 DynFuzzer 比 AFL 平均多发现 $0.26*\frac{10}{10+10+5}=10.4\%$ 的 crash。

5　结束语

针对现行模糊测试技术存在的种群基因收敛过早错误!未找到引用源。，路径覆盖率不足的问题提出了基于动态适应度函数的模糊测试方法，并基于该方法实现了模糊测试工具 DynFuzzer。通过实验证明 DynFuzzer 在模糊测试的路径覆盖率方面比 AFL 多发现 40%的路径，在发现 crash 方面平均比 AFL 多发现 10%的 crash。DynFuzzer 的不足有两个方面，一是在模糊测试前期产生新路径和 crash 的能力较弱；二是由于模块耦合度对控制流转化成马尔可夫链模型的影响，本方法对于复杂的非结构化程序和大型实际程序依然不能达到理想的效果，如何解决这两个问题还需要在接下来的工作中进一步研究。

参考文献：

[1] 廉美，邹燕燕，霍玮，等. 动态资源感知的并行化模糊测试框架 [J]. 计算机应用研究, 2017, 34 (1): 52-57. (Lian Mei, Zhou Yanyan, Huo Wei, etal. Parallel fuzzy testing framework for dynamic resource perception [J]. Application Research of Computers, 2017, 34 (1): 52-57.)

[2] 李红辉，齐佳，刘峰，等. 模糊测试技术研究 [J]. 中国科学：信息科学, 2014, 44 (10): 1305-1322. (Li Honghui, Qi Jia, Liu Feng, etal. Research on fuzzy testing technology [J]. Chinese science: information science, 2014, 44 (10): 1305-1322.)

[3] 吴志勇，王红川，孙乐昌，等. Fuzzing 技术综述 [J]. 计算机应用研究, 2010, 27 (3): 829-832. (Wu Zhiyong, Wang Hongchuan, Sun Lechang, etal. Review of fuzzing technology [J]. Application Research of Computers, 2010, 27 (3): 829-832.)

[4] Peach [EB/OL]. (2014-02-23) [2015-11-28]. http://community. peachfuzzer. com/.

[5] Luo Yongbiao, Ye Jun, Ma Xiaowen. Multicriteria fuzzy decision-making method based on weighted correlation coefficients under interval-valued intuitionistic fuzzy environment [J]. European Journal of Operational Research, 2010, 205 (1): 202-204.

[6] Vuori J. Student engagement: buzzword of fuzzword? [J]. Journal of Higher Education Policy & Management, 2014, 36 (5): 509-519.

[7] Caroline Lemieux, Koushik Sen. FairFuzz: targeting rare branches to rapidly increase greybox fuzz testing coverage [J]. 2017.

[8] Serebryany K. Continuous fuzzing with Libfuzzer and address sanitizer [C]// Cybersecurity Development. 2017: 157-157.

[9] Cha S, Woo M, Brumley D. Program-adaptive mutational fuzzing [C]// Proc of IEEE Symposium on Security and Privacy. 2015: 725-741.

[10] Rawat S, Jain V, Kumar A, *et al.* VUzzer: application-aware evolutionary fuzzing [C]// Proc of Network and Distributed System Security Symposium. 2017.

[11] Pham V T, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain [C]// Proc of ACM SIGSAC Conference on Computer and Communications Security. 2016: 1032-1043.

[12] Wang Junjie, Chen Bihuan, Wei Lei, *et al.* Skyfire: data-driven seed generation for fuzzing [C]// Security and Privacy. 2017: 579-594.

[13] Yoshida Y. Markov chains with a transition possibility measure and fuzzy dynamic programming [J]. Fuzzy Sets and Systems, 1994, 66 (1): 39-57.

[14] Mohamed M A, Gader P. Generalized hidden Markov models I: theoretical frameworks [M]. [S. l.] : IEEE Press, 2000.

[15] Yan S, Wang Ruoyu, Christopher S, *et al.* SOK: (State of) the art of war: offensive techniques in binary analysis [C]// Security and Privacy. 2016: 138-157.

[16] Cifuentes C, Hoermann C, Keynes N, *et al.* BegBunch: benchmarking for C bug detection tools [C]// Proc of International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM Sigsoft International Symposium on Software Testing and Analysis. New York: ACM Press, 2009: 16-20.

[17] 王丽薇，洪勇. 遗传算法的收敛性研究 [J]. 计算机学报, 1996, 19 (10): 794-797. (Wang Liwei, Hong Yong. Study on convergence of genetic algorithm [J]. Chinese Journal of Computers, 1996, 19 (10): 794-797.)